Elementos Avançados de C#

Propriedades

Programação tradicional usando C#

```
class Circle
{
    private int radius;

    public void SetRadius(int value)
       { radius = value; }

    public int GetRadius()
       { return radius; }
}


// Utilização:

Circle c = new Circle();

c.SetRadius(10);

Console.WriteLine(
          "Radius=" + c.GetRadius()  );
```

Programação C# simplificada

```
class Circle
{
    private int radius;

    public int Radius
    {
        set
        { radius = value; }
        get
        { return radius; }
    }
}

// Utilização:

Circle c = new Circle();

c.Radius = 10;

Console.WriteLine( "Radius=" + c.Radius  );
```

Em C# é introduzido o conceito de propriedade usado nos componentes. Uma propriedade é usada como se fosse um atributo público mas o acesso é na realidade feito através de métodos **get** e **set**. Para se ter acesso ao valor recebido por uma propriedade dentro da sua classe usa-se a palavra-chave **value** como se fosse uma variável. Esta *variável* contém o valor recebido.
É possível omitir o **set** ou o **get** de uma propriedade tornando-a assim, respetivamente, só de leitura ou só de escrita.
Uma propriedade pode ser declarada como abstrata ou como virtual e pode ser usada em interfaces.

Indexers

Programação tradicional usando C#

```
class Ecra
{
 private char[,] ecra = new char[25, 80];

 public char GetCharAt(int x, int y)
    { return ecra[x,y]; }

 public void SetCharAt(int x, int y, char c)
    { ecra[x,y] = c; }
}

Ecra e = new Ecra();
e.SetCharAt(2,3,'c');
Console.Write(e.GetCharAt(2, 3));
```

Programação C# simplificada

```
class Ecra
{
  private char[,] ecra = new char[25, 80];

  public char this[int x, int y]
  {
    get { return ecra[x,y]; }
    set { ecra[x,y] = value;}
  }
}

Ecra e = new Ecra();
e[2,3] = 'c';
Console.Write(e[2, 3]);
```

Em C# é possível ter a notação usada com *arrays* em classes normais através de *indexers*. Neste caso definem-se os *indexers* de uma forma semelhante às propriedades com um método **set** usado para colocar o elemento no *array* e um método **get** usado para obter um elemento do *array*.
O parâmetro usado no índice dos *indexers* pode ser de qualquer tipo e pode, inclusive, fornecer-se mais que um parâmetro como é feito em *arrays* multidimensionais. No exemplo mostra-se uma classe **Ecra** com um *indexer* contendo 2 índices correspondendo a coordenadas dentro do ecrã.
Tal como nas propriedades é possível omitir o bloco **get** ou o bloco **set.**

Delegates e Eventos

Programação tradicional usando C#	Programação C# simplificada

-----	`public class Stock`
	`{`
	`  ...`
	`  public void ProcessPriceChanged(decimal price)`
	`  {`
	`  ...`
	`  }`
	`}`
	
	`public delegate void Notify(decimal newPrice);`
	
	`Stock obs = new Stock();`
	`Notify n = new Notify(obs.ProcessPriceChanged);`
	`// também é possível:`
	`// Notify n = obs.ProcessPriceChanged;`
	
	`n( 123 ); // corre o método guardado`

Em C# existe o tipo **`delegate`** cujas variáveis guardam referências de métodos. Na definição deste tipo usa a assinatura do método que é guardado. O nome da variável é fornecido no lugar do nome do método usado na assinatura. No exemplo acima as variáveis do tipo **`delegate`** com o nome **`Notify`** podem guardar métodos que não tenham tipo de retorno e que recebam um valor do tipo **`decimal`** como argumento.
Um **`delegate`** pode guardar mais do que um método, podendo ser acrescentados métodos usando o operador **`+=`** e retirados usando o operador **`-=.`**
Para executar os métodos guardados num **`delegate`** basta usar o nome da variável como se fosse de um método. No exemplo acima para correr os métodos guardados na variável **`n`** (do tipo **`Notify`**) basta chamar o método **`n`** ou seja **`n(123)`** e é executado o método **`obs.ProcessPriceChanged(123)`**, o único guardado no **`delegate`**.
Os métodos guardados nos *delegates* podem retornar um valor mas, neste caso, apenas o valor retornado pelo último método que foi guardado no *delegate* é devolvido. Também no caso de um dos métodos gerar uma exceção, esta exceção propaga-se aos outros métodos não sendo executado nenhum dos métodos seguintes.

Programação tradicional usando C#	Programação C# simplificada

----	`// Simples`
	`public class MyEventNotifier`
	`{`
	`  public event Notify NotifyCallBack;`
	`}`
	
	`// Com controlo da adição e remoção de eventos`
	`public class MyEventNotifier`
	`{`
	`  public Notify notifyCallBack;`
	
	`  public event Notify NotifyCallBack`
	`  {`
	`  add`
	`  {`
	`  notifyCallBack += value;`
	`  }`
	`  remove`
	`  {`
	`  notifyCallBack -= value;`
	`  }`
	`  }`
	`}`

Para simplificar o uso de *delegates* o C# introduz o tipo **`event`** que cria um **`delegate`** na classe implicitamente. Neste caso, ao expor o **`event`**, que deve ser público, esconde o acesso ao **`delegate`** criado internamente, que é privado. A alternativa de expor o próprio **`delegate`** como uma variável pública violaria a encapsulação. A inscrição dos métodos no evento faz-se tal como para os **`delegates`** através da utilização dos operadores **`+=`** e **`-=`**.
Na prática um evento funciona como uma espécie de propriedade implícita em que a variável associada é criada automaticamente pelo compilador e é privada. Tal como nas propriedades em que se pode definir o código a executar quando se coloca lá um valor (bloco **`set { }`**) ou se lê um valor (bloco **`get { }`**), nos eventos é possível definir o código a executar quando se adiciona um método com o operador **`+=`** (bloco **`add { }`**) ou quando se remove um método com o operador **`-=`** (bloco **`remove { }`**).

Sintaxe de Inicialização de Objetos

Programação tradicional usando C#	Programação C# simplificada

`public class Point`
`{`
`  public int X { get; set; }`
`  public int Y { get; set; }`
`  public string Color { get; set; }`
 	
`  public Point(int x, int y)`
`  {`
`  X = x;`
`  Y = y;`
`  }`
 	
`  public Point() { }`
`  public void Display()`
`  {`
`  Console.WriteLine("({0},{1})",X,Y);`
`  }`
`}`	

`Point pt1 = new Point();`
`pt1.X = 10;`
`pt1.Y = 10;`
`pt1.Color = "blue";`
`pt1.Display();`
	`Point pt1 = new Point{X=10, Y=10, Color="blue"};`
	`pt1.Display();`

`Point pt2 = new Point(20, 20);`
`pt2.Color = "blue";`
`pt2.Display();`
	`Point pt2 = new Point(20, 20){Color="blue"};`
	`pt2.Display();`

	`List<Point> pts = new List<Point>()`
	`{`
	`  new Point{X=10, Y=10, Color="blue"},`
	`  new Point(20, 20){Color="blue"},`
	`  null`
	`};`

Em C# é possível recorrer a uma sintaxe de inicialização de objetos que simplifica o código. Neste caso para inicializar uma variável basta fornecer a seguir ao construtor, entre chavetas e separadas por vírgulas, as várias propriedades seguidas dos seus valores de inicialização. Também os parenteses do construtor podem ser omissos quando se tratar do construtor sem argumentos.
É possível ainda usar esta sintaxe com a inicialização de coleções como é mostrado no exemplo.

Variáveis de Tipo Implícito

Programação tradicional usando C#	Programação C# simplificada
`double val = 25.0;`	`var val = 25.0;`
`List<string> nomes = new List<string>();`	`var nomes = new List<string>();`

Em C# é possível utilizar a palavra reservada **var** em vez do tipo. O compilador C# encarrega-se de atribuir o tipo correto no momento da atribuição tendo em conta o tipo do valor atribuído. A utilização de **var** tem várias restrições importantes: A primeira é que apenas pode ser usada para variáveis locais, ou seja não pode ser usada em variáveis membro de uma classe (campos em C#), não pode ser usada em parâmetros de métodos, nem pode ser usada como tipo de retorno dum método. Outra restrição é que a variável tem de ser obrigatoriamente inicializada aquando da sua declaração e não pode receber nessa inicialização o valor **null**.

Como regra de boas práticas não é normalmente aconselhável usar a palavra **var** para criar variáveis normais porque pode dificultar a leitura e interpretação do código.

Tipos *nullable*

Programação tradicional usando C#	Programação C# simplificada

`static void Main(string[] args)`
`{`
`  Nullable<int> i = null;`
 	
`  if (i.HasValue)`
`  Console.WriteLine(i.Value); //ou (i)`
`  else`
`  Console.WriteLine("Null");`
`}`
	`static void Main(string[] args)`
	`{`
	`  int? i = null;`
	
	`  if (i.HasValue)`
	`  Console.WriteLine(i.Value); //ou (i)`
	`  else`
	`  Console.WriteLine("Null");`
	`}`

As variáveis dos tipos por valor (v*alue* types) em C# têm sempre que conter um valor, não podendo ser **`null`**. Por vezes é útil que não tenham qualquer valor, indicando que o seu valor está indefinido. Isto é possível recorrendo aos tipos *nullable* do C#. Para representar um tipo *nullable* acrescenta-se um ponto de interrogação a seguir ao tipo simples. Os tipos *nullable*, são na prática estruturas genéricas **`Nullable<T>`**, onde estão definidas as propriedades **`HasValue`** que indica se a variável tem um valor ou não, e **`Value`** que permite ir buscar o seu valor se este não for **`null`**.

Operadores de *null-coalescing*

Programação tradicional usando C#	Programação C# simplificada

`int? a = null;`
 	
`int b;`
`if( a == null )`
`  b = -1;`
`else`
`  b = a.Value;`
 	
`Console.WriteLine(b);  // output: -1`
	`int? a = null;`
	
	`int b = a ?? -1;`
	
	`Console.WriteLine(b);  // output: -1`

O operator **`??`** (*null-coalescing operator*) verifica o operando da sua esquerda, se este for diferente de **`null`** retorna o seu valor, caso contrário retorna o valor do operando que está à sua direita.

Programação tradicional usando C#	Programação C# simplificada

`List<int> numbers = null;`
 	
`if( numbers == null )`
`  numbers = new List<int>();`
	`List<int> numbers = null;`
	
	`numbers ??= new List<int>();`

O operator **`??=`** (*null-coalescing assignement operator*) verifica o operando da sua esquerda e apenas atribui o valor do operando à sua direita se o operador da esquerda for **`null`** .

Operador condicional `null`

Programação tradicional usando C#	Programação C# simplificada

`if(A != null)`
`  if(B != null )`
`  A.B.Do(C);`
 	
`Tipo X = null;`
`if(A != null)`
`  if(B[C] != null )`
`  Tipo X = B[C];`
	`A?.B?.Do(C);`
	
	`Tipo X = A?.B?[C];`

O operador **`?.`** (*null-conditional operator*) evita a utilização de um teste a **`null`** no acesso a um membro de um elemento. No exemplo mostrado, apenas é executado o método final se nenhum dos elementos anteriores for **`null`**.

Também existe um operador semelhante, o operador **`?[]`** que se aplica a elementos de um *array*.

Métodos Anónimos

Programação C# avançada

```
public delegate void Notify(decimal newPrice);

Notify n += delegate(decimal price ) {
       Console.WriteLine(">New price: " + price); };


n( 123 ); // corre o método anónimo
```

Em C# é possível definir métodos anónimos usando a palavra reservada **`delegate`** como nome do método e fornecendo os parâmetros e a definição do mesmo de seguida. No exemplo dado em vez de se adicionar ao evento um método de um objeto ou um método incluído na classe onde está o código fornece-se diretamente o método com parâmetros e código. Esta sintaxe simplifica o trabalho de criar e escrever métodos com poucas linhas de código que normalmente só seriam usados para a resposta a um evento dentro da mesma classe.

Expressões Lambda

Programação C# avançada

```
public delegate void Notify(decimal newPrice);

Notify n += price => Console.WriteLine(">New price: " + price);


n( 123 ); // corre a expressão lambda
```

Em C# é possível definir expressões lambda usando o operador **`=>`** . As **expressões lambda** podem ser usadas em todos os lugares em que se espera uma referência para um método (ou seja uma variável do tipo **`delegate`**).

A sintaxe das expressões lambda é muito simples:

Parâmetro(s) entrada => Instrução(ões) a executar

Com muita frequência o compilador consegue inferir o tipo do parâmetro de entrada com base no contexto onde está a ser usada a expressão. Caso contrário pode-se fornecer o tipo usando parênteses. No exemplo dado far-se-ia: **`(decimal price) => ...`** Também se podem fornecer várias instruções de código colocando-as entre chavetas: **`(decimal price) => { inst1; inst2; }`**. Finalmente se for necessário fornecer mais de um parâmetro basta colocá-los entre parênteses separados por vírgulas não sendo igualmente necessário colocar os seus tipos se o compilador os conseguir inferir. Por exemplo: **`(x, y, raio) => ... ou (int x, int y, double raio) => ...`**

Métodos de extensão

```
public static class MyExtensionMethods
{
    public static bool IsNumeric(this string s)
    {
        float output;
        return float.TryParse(s, out output);
    }
}

string test = "4.0";
if (test.IsNumeric())
    Console.WriteLine("Yes");
else
    Console.WriteLine("No");
```

Normalmente quando se pretende acrescentar um método a uma classe altera-se essa classe ou deriva-se uma classe nova que acrescenta esse método. Alterar a classe pode não ser possível se a classe fizer parte de uma biblioteca em que não se tem acesso ao código fonte. Da mesma forma derivar uma classe pode não ser possível se a classe estiver selada (ou se se estiver a usar uma estrutura). Os **métodos de extensão** permitem acrescentar métodos a uma classe sem que seja necessário usar qualquer das opções referidas. Para definir um método de extensão começa-se por criar uma classe estática onde será definido esse método (ou usar uma existente). De seguida na definição do método de extensão (que é obrigatoriamente **`static`**) deve-se colocar como primeiro parâmetro o nome da classe à qual se está a acrescentar o método antecedido pela palavra reservada **`this`**. Esta palavra reservada só pode aparecer aplicada ao primeiro parâmetro. Após colocar um identificador para este primeiro parâmetro, colocam-se a seguir os restantes parâmetros do método que serão aqueles que irão ser realmente usados. No exemplo acima acrescentou-se à classe **`string`** o método **`IsNumeric`** que não leva argumentos. Neste caso na definição deste método na classe **`MyExtensionsMethods`** apenas foi necessário colocar o primeiro parâmetro que identifica a classe **`string`** como sendo a classe que recebe o método.